A Distributed Task Queue for Load Balancing on the CM5 (Draft)

Chih-Po Wen
Computer Science Division
University of California, Berkeley
cpwen@cs.berkeley.edu

March 7, 1995

Abstract

We present a distributed task queue for dynamic load balancing on the CM5 multiprocessor. The task queue, though physically distributed among a group of processors, provides a shared programming abstraction for allocating and scheduling tasks. In addition to dynamic load balancing, the task queue also supports automatic termination detection. We discuss the functionality and interface of the task queue, sketch the design trade-offs, and describe the main modules of its implementation.

1 Introduction

For many applications, load balancing cannot be performed a prioi. A typical example of such applications is the search problem, where tasks are created at runtime based on the current status of computation. Also, the execution time of tasks may be unpredictable, even though the number of tasks and their dependency can be determined in advance. Dynamic load balancing must be used to deal with such irregularity.

A task queue is commonly used to implement dynamic load balancing. A task queue serves as a common repository for tasks from multiple processors. It is also responsible for scheduling tasks for execution when requested by the processors. The program starts with some initial tasks, which may generate more tasks when processed; it ends when the task queue is empty and all scheduled tasks have finished execution.

A naive implementation of a task queue is to make one processor responsible for all accesses. Such a *centralized* task queue will certainly become the performance bottleneck when the number of processors is large. A better implementation is to distribute the accesses among different processors. The advantage of a distributed task queue is two fold. First, the latency of the accesses can is reduced if they can be processed locally. Second, the bandwidth of accesses is increased if multiple accesses can be performed simultaneously on different portions of the task queue.

The interface of a distributed task queue resembles that of a sequential priority queue. However, some of the features requires new interpretations in the context of parallel programming. Among these are the meaning of priority and termination detection. We relax the strict priority semantics of a sequential queue to increase parallelism and to reduce the cost of implementation. We also augment the interface to handle termination detection correctly in the presence of multi-threading, which is unique to a parallel programming environment. Further discussion is given in Section 2

The most important function of a distributed task queue is to perform load balancing. We use a distributed protocol that balances load between nearest neighbors, which is extremely simple and works well in practice. The protocol also attempts to reduce the task migration penalty due to loss of locality. Section 3 describes the load balancing protocol in details.

The rest of the paper is organized as follows. Section 2 discuss the functionality and the interface of the distributed task queue. Section 3 describes the load balancing protocol. Section 4 describes the termination detection protocol. Section 5 sketches the implementation on the CM5 multiprocessor.

2 Functionality of the Distributed Task Queue

The design of the task queue is driven by the following two concerns: ease of use and high performance. In some cases, the user interface are augmented or relaxed so that the performance goals can be met. In this section we discuss these design trade-offs and give the detailed specification of the task queue interface.

2.1 Interface Description

The operations on a distributed task queue include: creating a task queue, enqueueing a task, dequeueing a task, and termination detection. The meaning of these operations are described below.

2.1.1 Creating Task Queues

A large application may use multiple task queues for storing different types of tasks, and each task queue may reside on a subset of all available processors. Therefore, when creating a task queue, the programmer must specify the range of processors on which the task queue can be accessed.

The programmer must also specify the size of a task when creating a task queue. Using a uniform task size allows more efficient implementation of accesses. Note that the structure of a "task" is not visible to the task queue; the task queue simply treats each task as an array of bytes, which is to be interpreted by the computational part of the program. We do recommend, however, that the programmer put the handle (pointer) of a task on the task queue instead of the putting the task itself (which can be quite large). This is because tasks may be migrated more than once by the task queue, and larger tasks cause more overhead for migration.

The programmer can also supply optional parameters to choose a particular configuration of the task queue. The current task queue implementation allows several configurations for load balancing purposes (details in Section 3).

The create operation must be called by all processors, one of which is then designated as the "manager" of that task queue for termination detection (details in Section 4). The create operation returns unique id as the handle of the task queue.

2.1.2 Enqueueing a Task

The enqueue operation places a task on the task queue. By default the task goes to the local partition of the task queue. However, the runtime system is free to migrate the task to other processors to improve load balance.

Along with the task, the programmer supplies the following information: the name of the task queue for which the task is enqueued, the scheduling priority (detailed later), an estimate of its computation cost, and an estimate of the migration penalty. The latter two are used by the load balancing protocol to select the tasks for migration. The task queue only migrate tasks for which the computation cost is greater than the migration penalty.

In some cases it is beneficial for the programmer to place the task at a specific processor. For example, the placement may implement a static load balancing scheme, which is then augmented by the dynamic load balancing scheme provided by the task queue. The placement may also correspond to the location of the data which can be different from the creater of the task. The task queue provides another enqueue operation which takes a destination processor [1]. Note that tasks enqueued in this fashion are still eligible for migration.

2.1.3 Dequeueing a task

The dequeue operation schedules a task for execution based on the user-supplied priority. However, priority is not enforced strictly; priority is only respected on a per processor basis. The task queue may schedule a low priority task on one processor before a high priority task on another processor.

Our interpretation of priority is very different from that of a sequential priority queue, in that we use priority as a performance related "hint", instead of as a strict synchronization constraint. Enforcing global priority is expensive in a distributed environment, as a dequeue operation would require communication to find the top priority task. Strict global priority also lowers parallelism since low priority tasks can not start before all higher priority tasks finish. Relaxing the priority constraint, however, may lead to more work for certain applications.

For certain applications, the termination of the program depends on the fact that the highest priority task be processed in finite time. Although the task queue does not enforce global priority, it still works in such cases – since the highest priority task must have the highest priority on some processor, it will be dequeued and executed in finite time.

The setting of priority involves the following tradeoffs. In general, it is advantageous to give high priority to tasks that create more tasks for parallelism. The increase in parallelism, however, may again lead to redundant work in branch-and-bound applications. Also, the more tasks there are, the more overhead their scheduling incurs. It seems desirable for the task queue to adapt its scheduling policy in response to the load of the system. We have not addressed this problem in our current implementation.

To invoke the dequeue operation, the programmer supplies the name of the task queue, a buffer to hold the dequeue task, and a buffer to hold general information on the task (such as priority and cost estimates). The primitive returns with a code indicating the status of dequeue: success, failure, or termination. A success code means that a task has been dequeue and placed in the user supplied buffer; a failure code means that no task is found, but some task may become available in the future; a termination code means that no task is found and no task will ever be placed in the task queue. In response to a falure code, the programmer

[1] This primitive must be used with care, as no flow control can be provided by the runtime system.

can spin on the dequeue primitive until a task is found, or switch to other computation to hide the latency of obtaining a task; in response to a termination code, the programmer enters the last phase of the program, which usually combines the results of different processors and terminates.

2.1.4 Termination Detection

Termination is detected by the dequeue operation via the status code. It can also be checked by a separate primitive which performs no dequeue operation. Note that the programmer does not have to invoke these operations to start the termination detection protocol; it is performed in the background by the task queue.

We must know what constitutes the condition for termination before we can understand its use. Termination is confirmed when there is no task being processed or stored in the *system*, where the system includes the task queue and the work space of the processors.

The emptiness of the task queue (which includes the local partitions and the network) can be checked by the task queue runtime system. The work space of the processors, however, can only be checked by the programmer. For example, the programmer may use multi-threading to hide latency, and the task queue has no way of knowing that tasks are buffered. To solve this problem, the programmer uses a special primitive whose sole purpose is to inform the task queue runtime system that a processor is free of tasks.

2.1.5 Polling the Task Queue

Ideally, all "background" operations such as load balancing and termination detection should be able to proceed without user intervention. However, this is difficult to do since we have only one thread of control per processor. Therefore, The programmer must poll the task queue explicitly to allocate processor time to the task queue runtime system. A poll simply means a transfer of control; the task queue decides what to do.

2.2 Summary of C Interface

Figure 1 shows the complete C interface of the task queue.

3 Load Balancing

In this section we describe the load balancing problem and outline our solution.

3.1 Design Issues

A dynamic load balancing protocol moves tasks between processors, in an attempt to minimize the overall execution time of the program. The dynamic load balancer must make the following decisions:

- When to initiate the load balancing protocol?
- From and to which processor to migrate tasks?
- What tasks to migrate?

```
/* Task Queue Interface */

/* Information about a task (returned by Taskq_dequeue). */
typedef struct {
        taskq* qid;
        int home_proc;                  /* home proc for storing sync bit    */
        int task_id;                    /* handle of the task                */
        int comp_cost;                  /* estimate of the computation cost  */
        int comm_cost;                  /* estimate of the communication cost*/
        int priority;                   /* the scheduling priority           */
} task_info;

/* Create a task queue and return its name. topology is MESH, RING,or HCUBE */
taskq* Taskq_create(int start_proc, int end_proc, int task_size, int topology);

/* Dequeue a task. Supply 0 for info to ignore task information. */
/*      Return code: 1 - success, 0 - failure, -1 - termination. */
int Taskq_dequeue(taskq* qid, task_info *info, void *data);

/* Enqueue a task. comp and comm are the cost estimates */
int Taskq_enqueue(taskq* qid, int priority, void *data, int comp, int comm);

/* Enqueue a task. Place the task on processor proc */
int Taskq_enqueueX(taskq* qid, int proc, int priority, void *data, int comp, int comm);

/* Check for termination. */
int Taskq_terminated(taskq* qid);

/* Signal that the local processor is free of task */
int Taskq_clear(taskq* qid);

/* Poll the task queue */
int Taskq_poll(taskq* qid);
```

Figure 1: The C Interface of the task queue.

The decision usually has to be made with incomplete information, since it is impossible to predict the future. In the following section, we summarize the common approaches to these problems and outline the trade-offs.

3.1.1 When to balance?

The load balancing protocol can be invoked by a busy processor (referred to as **busy-initiated**), or by an idle processor (referred to as **idle-initiated**).

Busy-initiated protocols are more efficient when tasks are about to be generated, that is, when there are many more idle processors than busy processors. Idle-initiated protocols are more efficient when the tasks are about to be exhausted, that is, when there are many more busy processors than idle processors. Busy-initiated protocols have the disadvantage of penalizing busy processors, because these processors observe the protocol overhead.

3.1.2 Where to balance?

Once the load balancing protocol is initiated, the requesting processor starts searching for candidate processors for task migration.

The selection process can be randomized, static, or adaptive. The randomized scheme picks an arbitrary set of processors; the static scheme select a set of neighbor processors that is determined a priori; the adaptive scheme uses global information collected at runtime to determine what processors to choose.

The randomized scheme sets up direct migration paths between processors. However, it may be inefficient in finding valid candidate processors. The static scheme is extremely simple, but may cause more overhead since tasks may be moved several times before it reaches its destination. The adaptive scheme performs best when the global information is accurate. However, there is a trade-off between the accuracy of global information and the overhead of collecting it.

3.1.3 What to Balance

After a busy processor is paired with an idle processor, their loads are re-distributed in order to reduce the overall execution time. In general, it is very difficult to determine the optimal distribution of work, since it also depends on the load of other processors.

A common approach is to migrate a fixed portion (less than 50%) of work from the busy processor. The amount of work is estimated by computation costs of the individual tasks, which must be specified by the programmer.

Task migration disturbs data locality, since local references made by the task may become remote after it is moved. To reduce the loss of data locality, the programmer specifies the penalty of migrating the tasks so that tasks with high penalty are kept local whenever possible.

Excessive task migration may congest the network or even overflow the local memory of the destination processor. Some form of flow control between the source and the destination processor must be enforced.

3.2 The Task Queue Load Balancing Protocol

We use an idle-initiated load balancing protocol for the distributed task queue. The processors are organized in a fixed topology (configurable by the user), and tasks move only between nearest neighbors. The task queue uses the computation cost estimate to balance work between processors, and in doing so it uses the migration penalty estimate to reduce the loss of locality.

The load balancing protocol proceeds as follows. Each processor communicates with its neighbors via a set of unidirectional channels with binary states (On and OFF). Migration to a neighbor processor is allowed if and only if the corresponding channel is ON. An idle processor initiates load balancing by turning on all OFF channels from its neighbors. Upon seeing an ON channel, a busy processor balance its with the neighbor processor, and then turns off the ON channel. A processor's state (idle or busy) is determined by the amount of work it is currently assigned – it is idle if its assigned work falls below a certain threshold. The protocol is completely distributed and no global information is required.

Flow control is enforced by having the idle processor set *channel* capacity of the channel, that is, the maximum number of tasks allowed through the channel, when it is turned on. The capacity corresponds to the amount of space reserved for incoming tasks from the channel. Unused space is freed by the busy processor when the channel is turned off.

A busy processor attempts to migrate a fixed portion of work to the requesting processor. The portion is controlled by a runtime variable, and is raised or lowered by the runtime system to balance the eveness of processor load and the overhead of task migration. Furthermore, a processor never becomes idle as a result of giving away tasks.

In the current implementation, the local partition of a task queue contains two queues. One of them contains tasks that can be migrated. The other stores tasks whose computation cost is no larger than the migration penalty; these tasks are not eligible for migration. Scheduling is performed according to the user-supplied priority.

The load balancing protocol has two important properties. First, the cost of obtaining a task is bounded, since each batch of requests (i.e., simultaneous channel-ON actions) acquires at least one task before the next batch can be sent. Second, there is a path of ON-channels from any idle processor to some busy processor. This guarantees that an idle processor will eventually obtain a task when the system is sufficiently loaded.

4 Termination Detection

In this section we describes the termination detection protocol of the distributed task queue.

4.1 Termination of a Single Task Queue

Here we consider the termination of one task queue.

Termination detection is invoked when all processors are likely to be idle. An inexpensive method for computing this hint is to count the number of processors that has become idle at least once since the last invokation of the termination detection protocol. Each processor owns a termination request token, which is passed to the manager processor of the task queue when the processor becomes idle. The manager invokes the termination detection protocol when all tokens have been received. If termination detection fails, the tokens are sent back to the processors and the process is repeated.

As stated in Section 2, termination detection requires checking the task queue, the network, and the processors' work space. In general, it is difficult to obtain a consistent view of a distributed system without taking a snapshot. Instead of blocking all processors when performing termination detection, we collect global information incrementally and then check for consistency.

A sufficient condition for termination (referred to as T) is when all processors are clear of work, and the total number of dequeues equals the total number of enqueues. The former checks the processor work space, and the latter ensures the emptiness of the local partitions and the network.

Instead of computing the termination condition (T) directly, we compute an approximate condition (T') asynchronously based on possibly stale processor information. However, we make sure that the approximation is safe, that is, T' is true implies T is true.

The procedure for computing T' is as follows. First, the manager sends a message to each processor to request for information. Upon seeing the message, each processor returns a triple (clear, enqueues, dequeues), where clear is true if and only if the processor is out of tasks (in the local queue and the work space). The processor must also disable task migration. The manager combines all triples received from the processors and compute the formula for T. If the condition is satisfied, the mananger sends a message to each processor to commit termination; otherwise, it sends messages to abort termination detection and resume task migration.

It remains to prove that the procedure yields a safe approximation to T. Suppose that T is false when T' is true. Then there must be an idle processor which receives a task after it responds to the request of the manager. The task must be sent by a processor before it is probed, when task migration is still allowed. As a result, the task is missed by the dequeue count. Since the dequeue count must equal the enqueue count for T' to be true, there must be a task which is missed by the enqueue count. This, however, implies that the task is enqueued to a processor after it is probed, and subsequently migrated to another processor not yet probed. The assumption contradicts with the protocol and the safety of T' is thus established.

To complete the correctness proof of T', we must show that T' becomes true eventually after T is true, that is, T' detects T. Note that once T becomes true it stays true. It is clear that T' must be true if the termination detection protocol is invoked after T becomes true. Since that fact that T is true sets up the condition for invoking the termination detection protocol, T' will eventually become true.

4.2 Termination of Multiple Task Queue

When multiple task queues are used in the same program, their dependence can be either of the following:

- Acyclic. The task queues forms a hierarchy, where adjacent levels of task queues act as producers and consumers of tasks. Termination detection can proceed top-down, since tasks the producers of tasks for a task queue are at the same or higher levels. To chain together the termination detection protocols, the programmer asserts that a processor is clear of task for task queue q only after all task queues higher than q have terminated. The program terminates when all task queues have terminated.

- Cyclic. The dependency graph for the task queues are no longer acyclic. The task queue runtime system must consider these task queues as a whole when performing termination detection; the local partitions of different task queues are grouped into one conceptual partition, and the termination detection protocol must combine the enqueue and dequeue counts of all task queues. The current implementation does not support such grouping.

5 Implementation Notes

The task queue runtime system is written in C with CMAML, the CMMD active message layer for the CM5.

To simplify programming, the task queue runtime system is written as a collection of atomic handlers invoked by remote processors. Therefore, all CMMD and CMAML interrupts must be disabled.